Train sklearn 100x faster

Evan Harris
Aug 29, 2019 · 7 min read ★

At Ibotta we train a lot of machine learning models. These models power our recommendation system, search engine, pricing optimization engine, data quality, and more. They make predictions for millions of users as they interact with our mobile app.

While we do much of our data processing with Spark, our preferred machine learning framework is scikit-learn. As compute gets cheaper and time to market for machine learning solutions becomes more critical, we've explored options for speeding up model training. One of those solutions is to combine elements from Spark and scikit-learn into our own hybrid solution.

Introducing sk-dist

We are excited to announce the launch of our open source project sk-dist. The goal of the project is to provide a general framework for distributing scikit-learn meta-estimators with Spark. Examples of meta-estimators are decision tree ensembles (random forests and extra randomized trees), hyperparameter tuners (grid search and randomized search) and multi-class techniques (one vs rest and one vs one).

Our primary motivation is to fill a void in the space of options for distributing traditional machine learning models. Outside of the space of neural networks and deep learning, we find that much of our compute time for training models is not spent on training a single model on a single dataset. The bulk of time is spent training multiple iterations of a model on multiple iterations of a dataset using meta-estimators like grid search or ensembles.

Example

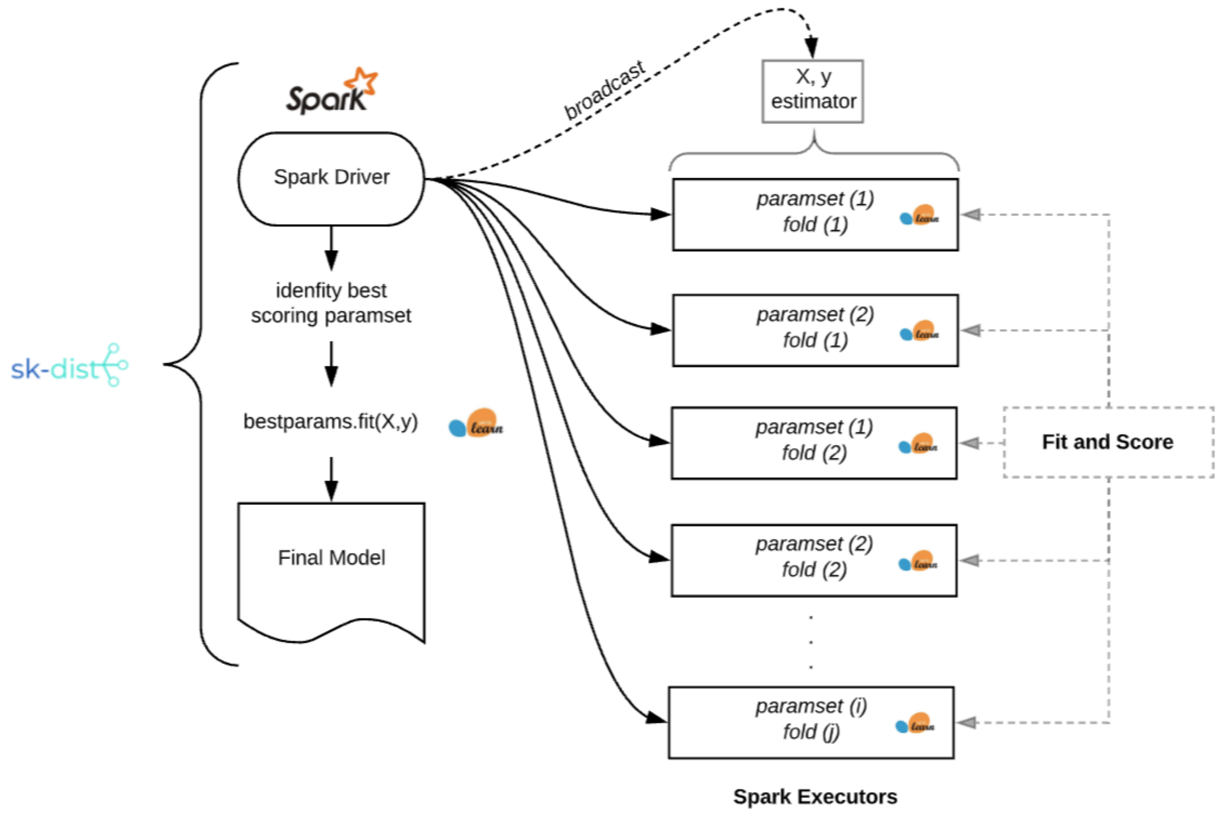
Consider the hand written digits dataset. Here we have encoded images of hand written digits to be classified appropriately. We can train a support vector machine on this dataset of 1797 records very quickly on a single machine. It takes under a second. But hyperparameter tuning requires a number of training jobs on different subsets of the training data.

Shown below, we've built a parameter grid totaling 1050 required training jobs. It takes only 3.4 seconds with sk-dist on a Spark cluster with over a hundred cores. The total task time of this job is 7.2 minutes, meaning it would take this long to train on a single machine with no parallelization.

```
import time

from sklearn import datasets, svm
from skdist.distribute.search import DistGridSearchCV
from pyspark.sql import SparkSession

# instantiate spark session
spark = (
    SparkSession
    .builder
    .getOrCreate()
    )
sc = spark.sparkContext

# the digits dataset
digits = datasets.load_digits()
X = digits["data"]
y = digits["target"]

# create a classifier: a support vector classifier
classifier = svm.SVC()
param_grid = {
    "C": [0.01, 0.01, 0.1, 1.0, 10.0, 20.0, 50.0],
    "gamma": ["scale", "auto", 0.001, 0.01, 0.1],
    "kernel": ["rbf", "poly", "sigmoid"]
    }
```

```
scoring = "f1_weighted"
cv = 10

# hyperparameter optimization
start = time.time()
model = DistGridSearchCV(
    classifier, param_grid,
    sc=sc, cv=cv, scoring=scoring,
    verbose=True
    )
model.fit(X,y)
print("Train time: {0}".format(time.time() - start))
print("Best score: {0}".format(model.best_score_))

------------------------------
Spark context found; running with spark
Fitting 10 folds for each of 105 candidates, totalling 1050 fits
Train time: 3.380601406097412
Best score: 0.981450024203508
```

This example illustrates the common scenario where fitting data into memory and training a single classifier is trivial but the number of required fits for hyperparameter tuning adds up quickly. Here is a look under the hood at running a grid search problem like the above example with sk-dist:



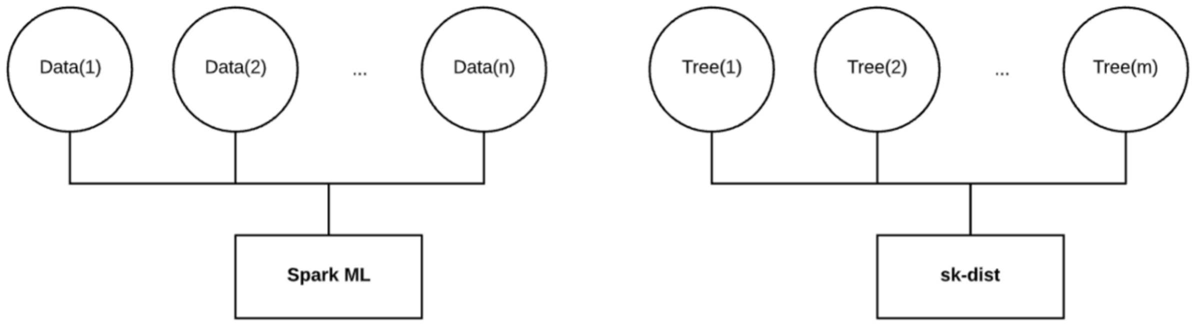
Grid Search with sk-dist

For real-world applications of traditional machine learning at Ibotta, we often find ourselves in a similar situation: small to medium sized data (100k to 1M records) with many iterations of simple classifiers to fit for hyperparameter tuning, ensembles and multi-class solutions.

Existing Solutions

There are existing solutions to distributing traditional machine learning meta-estimator training. The first is the simplest: scikit-learn's built-in parallelization of meta-estimators using joblib. This operates very similarly to sk-dist, except for one major constraint: performance is limited to the resources of any one machine. Even versus a theoretical single machine with hundreds of cores, Spark still has advantages like fine tuned memory specification for executors, fault tolerance, as well as cost control options like using spot instances for worker nodes.

Another existing solution is Spark ML. This is Spark's native machine learning library, supporting many of the same algorithms as scikit-learn for classification and regression problems. It also has meta-estimators like tree ensembles and grid search, as well as support for multi-class problems. While this sounds like it may be a sufficient solution for distributing scikit-learn style machine learning workloads, it distributes training in a way that doesn't solve the kind of parallelism of interest to us.



Distributed on Different Dimensions

As shown above, Spark ML will train a single model on data that is distributed on many executors. This works well when data is large and won't fit into memory on a single machine. However, when data is small this can *underperform* scikit-learn on a single machine. Furthermore, when training a random forest for example, Spark ML trains each decision tree in sequence. Wall time for this job will scale linearly with the number of decision trees regardless of the resources allocated to the job.

For grid search, Spark ML does implement a parallelism parameter that will train individual models in parallel. However each individual model is still training on data that is distributed across executors. The total parallelism of the job is a fraction of what it could be if distributed purely along the dimension of models rather than data.

Ultimately, we want to distribute our training on a different dimension than that of Spark ML. When using small or medium data, fitting the data into memory is not a problem. For the random forest example, we want to broadcast the training data in full to each executor, fit an independent decision tree on each executor, and bring those fitted decision trees back to the driver to assemble a random forest. Distributing along this dimension can be orders of magnitude faster than distributing the data and training decision trees in serial. This behavior is similar for other meta-estimator techniques like grid search and multi-class.